Olympiads in Informatics, 2015, Vol. 9, 75–88
2015 *Vilnius University*, *IOI*
DOI: http://dx.doi.org/10.15388/ioi.2015.07

Metamorphic Testing and DSL for Test Cases & Checker Generators

Ryan Ignatius HADIWIJAYA, M. M. Inggriani LIEM
Data and Software Engineering Research, School of Electrical Engineering and Informatics
Institut Teknologi Bandung
e-mail: ryan.ign54@gmail.com, inge@informatika.org

Abstract. In programming competition, a problem setter must prepare a task description, program solution, test cases and sometimes a checker. Test cases should be able to capture all possible cases; therefore, its preparation is time-consuming. Metamorphic Testing (MT) is a property-based testing method where relationships are defined between input and output to alleviate a test oracle problem. The success of MT relies on the existence of a Metamorphic Relation which is comprised of two interrelated relations: the test-case relation and the test-result relation. MR can be used for automated test-case generation and verification of results. In this research, we defined a Domain Specific Language (DSL) to describe metamorphic relations that will be used for test case and checker generation of programming tasks. Our method has been tested for tasks with Knapsack, Greedy, and Dynamic Programming solutions, and it has been proven, reliable, reusable and more systematic.

Keywords: test case generation, programming task checker, programming competition, Knapsack problem, Greedy problem, metamorphic testing.

1. Background

In Indonesia, autograder systems are used for national training programs and for the selection of IOI participants. To prepare a contest or training session, we have to define a problem set, which includes a description of the task, program solutions, and test cases (input as well as output). Some tasks also need checker. Up until now, the preparation has been carried out manually by a problem setter, including preparation of input and output test cases. Manual test case preparation is time-consuming and nearly impossible for a complex problem with a large amount of data. Therefore, test cases are generated by programs written one by one in a manner, specific to each problem set.

In IOI, there are two types of tasks, namely batch tasks and interactive tasks. In this paper, we focus on the batch task, which is judged by a grader and based on black-box testing. However, grading is more than testing because the grader must judge and give a score for each subtask that refers to a contestant's solution. A batch task has one or more subtasks, which in turn have constraints and scores. A good black-box testing method

uses all of the values in the input domain as input test cases. This would be impossible if the input data domain had very large (or even infinite) values. The problem setter must select reasonable values to be used in grading. This problem is known as the test-case selection strategy. If test cases are selected manually, either intuitively, or randomly, then their coverage is not guaranteed. The programming of a task also has time constraints that require appropriate test cases. An incorrect solution can be judged as an acceptable answer when the test cases do not precisely reflect the conditions. On the other hand, a redundant test cases will consume unnecessary time of execution and consume CPU resources. Good test cases must have sufficient test coverage and reasonable quantity and properties. Therefore, a test-case specification is needed. We aim to write a test-case generator based on specifications so that it is self-documented, and the scientific committee can verify its coverage and quality.

Some tasks may have many possible solutions. In order to optimize the autograding process, the team writes a checker instead of generating all possible solutions. The checker is used to compare contestant output to output test cases. Usually, the checker is also made ad-hoc and by writing a specific program for a specific task. It is difficult to verify its correctness. A faulty checker can make an incorrect answer become acceptable. In our research, we aim to provide the problem setter with a checker specification.

Before using the system explained in this paper, test cases for Indonesian training programs are actually being generated by the program as much as possible and not completely manual. However, test-case generation source code depends on problem setter and not driven by specific method. Checkers are programmed one by one specifically for each task and are not generated. Metamorphic Testing has the potential to be implemented as a method for improving test case and checker generation which implies an improvement in the automatic grading process.

2. Related Works

Our work is inspired by Metamorphic Testing (MT) (Chen *et al.*, 1998; Chen *et al.* 2004; Zhou *et al.*, 2004; Mahmuda *et al.*, 2011; Barus *et al.*, 2011) and Domain Specific Language (DSL) (Im *et al.*, 2008; Ghosh, 2011). Chen *et al.* (1998) which suggest using Metamorphic Testing for test case generation. Test case generation based on Metamorphic Relation (MR) could be automated (Gotlieb and Botella, 2003), with the MR coded directly in a general programming language. In our approach, we generate test cases (input, output) and a checker for a programming task by defining a Domain Specific Language for representing MR and input/output.

2.1. *Metamorphic Testing and Metamorphic Relation*

Metamorphic Testing (MT) is a technique to generate follow-up test cases based on existing test cases that have not revealed any failure. MT generates follow-up test cases by making reference to the metamorphic relation (MR).

An MR refers to two types of relations. First, by referring to the MR of the target function, follow-up test cases can be automatically constructed, executed, and checked to further verify the program. Metamorphic testing is to be used in conjunction with a test-case selection strategy S. Test set T generated from S must also exist in the first place.

Second, MR refers to the verification of testing the output (test result). Suppose we have a metamorphic relation R of function *f*, of which p is an implementation. The second relation refers to necessary properties of the target function *f* where if any of these properties does not hold, then program p is faulty. Metamorphic testing makes use of the relationship between the inputs and outputs, and involves multiple executions of p.

For example, if $f(a) = e^a$, then the property $e^a \times e^{-a} = 1$ is a typical MR. For a test case $a = 0.3$, metamorphic testing generates its follow up test case $a' = -0.3$ and then runs the program again on a'. The relation of the two outputs is checked against the expected relation $p(0.3) \times p(-0.3) = 1$. If this identity does not hold, then a failure is immediately detected (Zhou *et al.*, 2004). Another example of trivial MR is $\sin x = \sin(\pi - x)$ for a program that computes $\sin(x)$.

In testing, successful test cases are test cases which do not reveal any failure of the program. In a contest, successful test cases are test cases that reveal a correct answer and give a full score. Therefore, successful test cases have been considered useless in conventional testing because they do not reveal any failures. In other words, in a conventional testing, the successful test cases are discarded or retained. In contrast, metamorphic testing can be employed to make use of the successful test cases. In the context of the programming task, this idea will be used for validating the result of the generator (test input), and to accept or reject a generated test case.

Another example of follow-up test cases is illustrated in Fig. 1 (Murphy, 2010). Fig. 1 illustrates an example of a metamorphic relation to sum all elements of an unsorted numerical array. Permute, add, multiply, include and exclude are five examples of metamorphic relation. Five sets of new test cases can be generated based on an initial

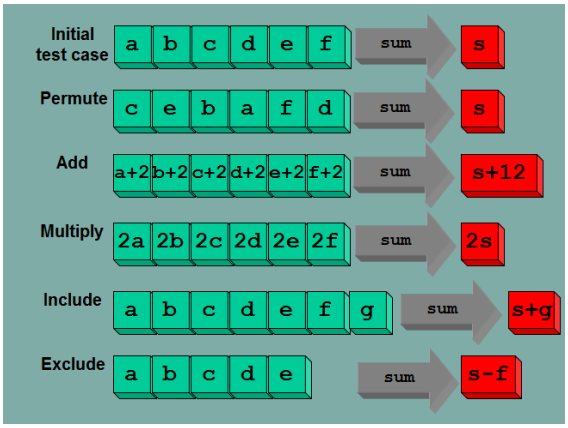

Fig. 1. Example of Metamorphic Relation for Sum Function (Murphy, 2010).

successful test case in order to reveal faults in the program. The output of these new test cases can be determined easily by its metamorphic relation, and this can save time and reduce the cost of making test cases.

2.2. *Domain Specific Language*

A DSL is a programming language that is targeted for a specific domain. It contains syntax and semantics that models concepts at the same level as abstraction of the problem domain.

Compared to GPL (General Purpose Language), DSL is shorter and simpler (Ghosh, 2011). DSL is easier to understand by domain experts. By using DSL, users can focus on the problem and deliberate from detail implementation. DSL is designed to be used intuitively.

A domain-specific language is created specifically to solve problems in a particular domain and is not intended to be able to solve problems outside that domain (although it may technically be possible). DSL for the business domain is defined to externalize business rules and computations, such as tax calculations, salary calculations, or financial engineering.

Examples of domain-specific languages include HTML and SQL for relational database queries, YACC grammars for creating parsers, regular expressions for specifying lexers, Csound for sound and music synthesis, and the input languages of GraphViz and GrGen, software packages used for graph layout and graph rewriting.

DSL is also used in automated test case generation (Im *et al.*, 2008). We intend to define a specific DSL to solve the generation of test cases of a programming task, based on MR.

3. Problem Statement and Objectives

When test cases are generated randomly, the coverage is not guaranteed and the generator is not reusable. More than that, its documentation is not preserved. Test case generation that contains initial specification can solve this problem. DSL offers precise and simple expressions well known by experts of the domain. Specifications written in a DSL will preserve the documentation of test cases and the checker. MT is property-based testing and provides a method for automated generation of test cases by defining MR. MT will improve the quality of test cases so that the autograding process will be more robust. During the grading process, the checker must not only check the properties of the output, but also checks the relations of many executions.

In this research, we combine the idea of Metamorphic Testing by defining the Metamorphic Relation with a Domain Specific Language. We aim to deliver a solution to a problem setter, so that the problem setter as the domain expert can express test cases and a checker by a specification written in a DSL.

The advantages of our approach are :

1. The problem setter focuses on specifications rather than on a program.
2. The system provides a reusable library of common test cases and checkers, since algorithmic solutions can be grouped by a variety of techniques such as Knapsack, Greedy, Dynamic Programming, Geometry, etc. It uses standard data structures (arrays, trees, graphs, etc.) since each techniques and data structure has a common MR.

4. Proposed Solution

First of all, we define a DSL grammar to represent MR, input/output variable names, constraints and their values, input/output format, and checker expression. A part of the DSL grammar represents the name of the class, and its features and six main declarations are presented in Code 1. The complete grammar is accessible in `https://github.com/ryanignatius/CheckerDSL/tree/master/Grammar`. Our system will read the specification, and generate test cases and checkers. The problem setter is not required to write a program, compared to the usage of a framework or an existing library (Mirzayanov, 2008).

```
Class:
    'class' name=ValidID '{' features+=Feature* '}';
Feature:
    ChkVariableDeclaration | Method | Format | Check | MR | Score;
ChkVariableDeclaration:
    type=ChkTypeReference ('[' sz+=CHK_NUMBER ']')* name=ValidID
      ('(' limit1=Limit (';' limit+=Limit)* ')')?
      ('value' '{' spValue=SpValue '}')?;
Method:
    'op' type=JvmTypeReference name=ValidID
      '('(params+=FullJvmFormalParameter
        (',' params+=FullJvmFormalParameter)*)?')'
       body=XBlockExpression;
Format:
    InputFormat | OutputFormat;
Check:
    check='check' '{' ( chk+=(ChkExpression | ChkLoopExpression) )* '}';
MR:
    mr='MR' num=INT '{'
        (mrExp+=(ChkExpression | ChkLoopExpression))*
        followup=FollowUp
        property=Property
        '}';
Score:
    'score' '{' (scores+=ChkScoreExpression)+ '}';
```

Code 1. A Part of DSL Grammar.

In the auto-grading process, a grader executes a contestant's program with a corresponding input test case, and then compares the execution result with the output test cases. If the output of the contestant's program is equal to output test cases, then the

grader judges it as a correct answer. For some tasks, contestant outputs are checked by provided checker(s). The contestant obtains a score for each correct input-output set and the final score for a task is visualized on a scoreboard. In our case, the checker does not only check a single-run output. The checker checks MR and other properties defined in the specification.

By using MR, the relation between the output of one run to another run (related by MR) can be checked. This will increase test robustness. When a relation between two outputs does not conform to the defined MR, then the grader will judge it as a wrong answer. We define each MR as a checker. A checker is a predicate that can check whether a set of output corresponds to a predefined MR, simply checks the property of the output, or checks the coverage of the input.

The problem setter must write a problem solution, a base-test case (a set of minimum test cases), and a specification file. The specifications are written in the DSL and consist of six declarations:

1. **Variable declaration**. The problem setter declares variable names, variable constraints, and test-case domain partitions that will be used in other sections. A variable can be declared as a JAVA primitive type, an array or a specific data structure such as a graph, tree or list. Each variable is generated as a private attribute in the generated JAVA file. For each attribute, functions are also generated to read, write, and validate.
2. **Input/output format declaration**. The problem setter declares the input and output formats, where values of variables will be read or written. The system will generate functions to read and write all variables that have been declared in the previous section. The read/write function will validate the input or check the output based on the constraints that have been defined in the variable declaration section.
3. **Predicate declaration**. A predicate is a function returning a boolean. This predicate will be used to generate a checker. If the input or output to be checked passes all tests by invoking the predicates, then the output will be judged as a correct answer.
4. **MR declaration** consists of follow-up and properties. Follow-up will be generated based on the MR, and properties are used to ensure that MR is satisfied.
5. **Other function declarations**. The problem setter can define specific functions. The system provides predefined functions such as *sort, swap, min, max, check if a number is a prime number*, etc. If a function is not defined in the library, the problem setter must implement it. The problem setter can enrich his environment by registering his function in the library.
6. **Score declaration**. The problem setter defines the score distribution for each subtask. In IOI, the score is given when a contestant's program passes all test cases in the subtask.

Test case and checker generation are described by the work flow depicted in Fig. 2. The first phase of the process consists of two parts that can be carried out in any order. The first part is processing the DSL specification file within the XText framework (Xtext, 2014) and produce a file named `GeneratedClass.java`. This file is then compiled with the given `MainGenerator.java` and `LibraryFunction.java`. `Main-`

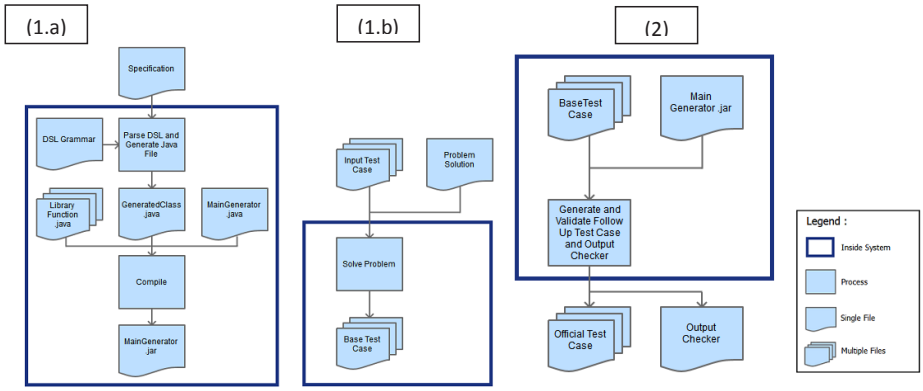

Fig. 2. Workflow of Test Case and Checker Generation.

`Generator.java` is a main program that receives parameters from the problem setter (output checker, input-output test cases, minimum number of test cases, mapping of test cases to subtasks). `LibraryFunction.java` contains predefined functions, that can grow as we may find other generic functions in the future. The result of this compilation is a jar file that will be used in the second phase. The second part is the generation of base test cases by running the problem solution with the given input.

The second phase is the generation of input files, output files, score files and checkers. The generation process is repeated and for each generation the program will validate input and output (defined in the specification). The system will reject test cases that do not comply with the specification, and repeat the process until the problem setter obtains sufficient test cases described in the specification. For each MR and given test case, the system will generate a follow-up test case. A corresponding checker will also be generated for checking the MR of the given test case output with generated follow-up test-case outputs.

5. System Architecture

The user of our system is the problem setter. The architecture of the system consists of four layers as can be seen in Fig. 3. The first layer is Java, containing JVM as the runtime environment. The second layer is IDE, whose framework contains Eclipse and XText, running on JVM. The DSL grammar, JVM Model Inferrer, Library Function, and Main Generator are put in the Developer layer. This layer is provided by us. On top of the third layer is the user layer, where a problem setter defines the specification and obtains generated classes. The problem setter interacts with the system through components in this layer. For each task, the problem setter writes a module. Checker specification is defined by a user using the XText component. Checker specifications are parsed by the XText parser using DSL grammar. If parsing is successful, then a checker will be generated by XText based on the existing JVM Model Inferrer. This generated file will be compiled

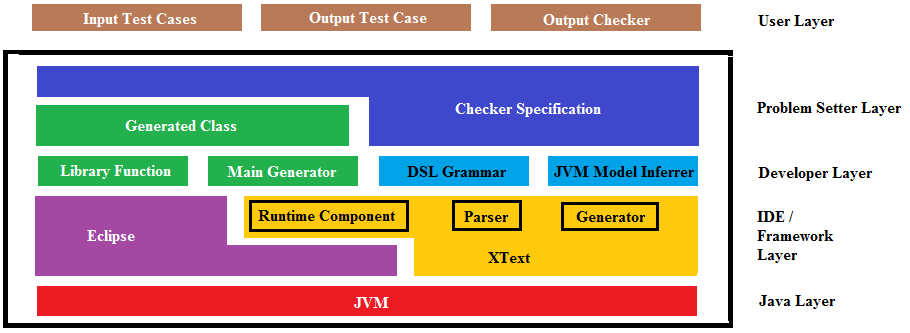

Fig. 3. System Architecture.

by the Library Function and Main Generator. The execution of these files will produce files (test-case input, test-case output, score, and checker).

DSL grammar is implemented using the XText framework, a plugin for the Eclipse IDE. Eclipse's features such as autocomplete and automatic error checking are also available while the problem setter defines the specifications. These specifications will be translated into a `.java` file. XText is used because of its availability as part of the Eclipse. Eclipse is a cross platform IDE, independent of a specific Operating System. Xtext is also integrated with JAVA so that our DSL can take advantage of the existing JAVA data type.

Some metamorphic relations are common in mathematical functions (Murphy, 2010), such as Additive (increases or decreases numerical values by a constant), Multiplicative (multiplies numerical values by a constant), Permutative (permutes the order of elements in a set), Invertive (takes the inverse of each element in a set), Inclusive (adds a new element to a set), Exclusive (removes an element from a set), and Compositional (creates a set from a number of smaller sets). These relations are generally applicable to tasks that deal with numerical inputs and outputs. Since many tasks in IOI involve numerical input and output, these relations are frequently applied. Therefore, we also have implemented these relations in our Library Function as reusable MR.

6. Case Studies

We applied the methods and tools to generate test cases for tasks with Knapsack, Greedy and DP solutions. More than that, we also demonstrate that the generic knapsack MRs can be used as a reusable specification for a more specific knapsack problem.

6.1. *Knapsack*

The Knapsack program accepts three sets of integers. Two n-tuple sets, $P = \{p_1, p_2, \ldots, p_n\}$ and $W = \{w_1, w_2, \ldots, w_n\}$ represent the profits and the weights of n items,

respectively; while another m-tuple set $C = \{c_1, c_2, \ldots, c_m\}$ contains the capacities of m knapsacks. The outputs of Knapsack are one n-tuple set $Y = \{y_1, y_2, \ldots, y_n\}$ and one positive integer TP. $y_i = j$ (where $i = 1, 2, \ldots, n$ and $j = 0, 1, \ldots, m$) states that the i^{th} item should be put into the j^{th} knapsack. If $y_i = 0$, it means that the i^{th} item will not be selected into any knapsack. TP represents the total profit of the picked items. The Knapsack program attempts to calculate the optimal solution and thus to maximize the total profit. (Mahmuda *et al.*, 2011).

For generic knapsack problem, we adopted 10 Metamorphic Relations defined by Mahmuda (Mahmuda *et al.*, 2011) and translated into 10 MR declarations, MR1 to MR10. These MRs will be used to generate input test cases and checkers to check the relation between outputs. Examples of MR1 and MR5 are translated into DSL expressions :

1. **MR1**: Swap the k^{th} and the l^{th} items, where $1 \leq k < l \leq n$, and $p_k \neq p_l$ or $w_k \neq w_l$. We can get the follow-up test case $T' = \{P', W', C\}$, where $P' = \{p_1, p_2, \ldots, p_l, \ldots, p_k, \ldots, p_n\}$ and $W' = \{w_1, w_2, \ldots, w_l, \ldots, w_k, \ldots, w_n\}$. The output corresponding to T' is $O' = \{Y', TP'\}$. We should have $Y' = \{y_1, y_2, \ldots, y_l, \ldots, y_k, \ldots, y_n\}$ and $TP' = TP$.

   MR1 expressed in DSL :

```
MR 1 {
  (select(k,l) where 1<=k and k<l and l<=n and p[k]!=p[l]
      or w[k]!=w[l])
  followup {
    (p' = swap(p,k,l))
    (w' = swap(w,k,l))
  }
  check {
    (y' = swap(y,k,l))
  }
}
```

2. **MR5**: Change the capacity of the 1^{st} knapsack to a new value c_1', where c_1' is equal to the sum of the weights of all items put into the 1^{st} knapsack. We can get the follow-up test case $T' = \{P, W, C'\}$ where $C' = \{c_1', c_2, \ldots, c_m\}$. The output corresponding to T' is $O' = \{Y', TP'\}$. We should have $Y' = Y$ and $TP' = TP$.

   MR5 expressed in DSL :

```
MR 5 {
  (def c1 = sum(w) where y[i]==1)
  followup {
    (c'[1] = c1)
  }
  check {
    }
}
```

This problem has multiple values that can produce an optimal total profit. Therefore, we have to define a checker to verify the correctness of a solution. A checker (source code) will be generated from the specification to check the following properties :

1. The total profit of the output produced must be equal to the total profit generated in the answer.
2. The sum of all profits of the item must be equal to the total profit.
3. The sum of weight of all items in each knapsack must be less than or equal to the capacity of the corresponding knapsack.

MR1 to MR10 will be used in follow-up test-case generation. We give an illustration of test-case generation for MR1 and MR5 in Table 1.

Complete implementation of the DSL specification of the Knapsack problem, all generated input, output, and checkers are accessible in:

`https://github.com/ryanignatius/CheckerDSL/tree/master/Examples/Knapsack/Knapsack1`.

6.2. *Specific Knapsack*

From a knapsack case study, we can see that MR1 to MR10 can be used for other tasks with a knapsack solution, for example "Polo the Penguin and The Test" (`http://www.codechef.com/problems/PPTEST`). Here is an example of how test cases and checkers from the knapsack case study can be reused for another task that has the nature of a knapsack problem. In this task, there is one knapsack. The amount of time represents the capacity of the knapsack. Tests represent the items that must be put in the

Table 1

Test-case generation example for MR1 and MR5 of a Knapsack problem

Test Case	File Input	File Output	Explanation
Original Test Case	3 2 5 4 8 2 3 5 6 1	1 1 0 9	Input and Output are defined by a problem setter. Input: 1st line: 3 items to be put in 2 knapsacks 2nd line: profit of each item 3rd line: weight of each item 4th line: capacity of each knapsack Output : 1st line: knapsack number for each item Total Profit = 9
MR1	3 2 5 8 4 2 5 3 6 1	1 0 1 9	File input and output are generated based on the original test case and MR1 (by swapping the 2nd and 3rd item) Output: Total profit = 9
MR5	3 2 5 4 8 2 3 5 5 1	1 1 0 9	File input and output are generated based on the original test case and MR5 (by changing the capacity of the 1st knapsack to the sum of the weights of all items put into the 1st knapsack)

knapsack. Profit is analogous to the number of tests contain this question (C[i]) multiplied by the number of points of this question (P[i]). However, MR10 is not applicable since the number of knapsacks is one. We replace MR2 (to add profit to an item) by MR11 and MR12. MR 11 is to add the number of tests (C[i]) to an item. MR12 is to add the number of points to an item (P[i]). We also remove the variable y, since the task asks for the total profit only.

We generate test cases and checker for another example ("farmer"), is taken from IOI 2004 task (`http://www.ioinformatics.org/locations/ioi04/contest/day2.shtml#p2`). This task can be modeled as a knapsack problem, so we can use the same MRs of the knapsack problem to generate test cases and checkers for this task. In this task, there is one knapsack. The number of cypress trees to be selected represents the capacity of the knapsack. Fields and strips represent the items. The number of trees in each field represents a profit and weight of the item. The number of trees in each strip represents weight for the item and the profit for this item equals to the weight of this item minus one. MR10 is not applicable since the number of the knapsack is one. We replaced MR1 (to swap two items) by MR11 and MR12. MR11 is to swap two fields and MR12 is to swap two strips. We replaced MR6 (to add a new item) by MR13 and MR14. MR13 is to add a new field and MR14 is to add a new strip. We replaced MR7 (to delete an item) by MR15 and MR16. MR15 is to delete a field and MR16 is to delete a strip. MR3 and MR4 are not applicable to this task since the weight and profit of an item can't be manipulated individually.

In this case study, we have demonstrated the reusability of our system and how to modify an existing metamorphic relation for a variant of a Knapsack problem.

Detailed implementation of DSL specification for this Knapsack problem, the generated input, output and checker are accessible here:

`https://github.com/ryanignatius/CheckerDSL/tree/master/Examples/Knapsack/Knapsack2`

and

`https://github.com/ryanignatius/CheckerDSL/tree/master/Examples/IOI%20Task/farmer`.

6.3. *Greedy*

The Greedy program (key-lock problem) receives input that is a set of keys $K = \{k_1, k_2, \ldots, k_x\}$ and a set of locks $L = \{l_1, l_2, \ldots, l_y\}$, where $x, y > 0$. For every pair (k_m, l_n), we define $r(m, n)$ as a relationship between key k_m and lock l_n such that $r(m, n) = 1$ if k_m opens lock l_n and $r(m, n) = 0$, otherwise. (Barus *et al.*, 2011)

We adopted nine Metamorphic Relations defined for this problem from Barus (Barus *et al.*, 2011). This problem does not need a checker, therefore the checker session is "NONE". Examples of MR and DSL expressions for Greedy problems are given as follows.

1. **MR3**: Adds an insecure lock column

MR3 expression in DSL :

```
MR 3 {
  followup {
    (y' = y+1)
    (m' = addColumn(m))
    for (i,x){
      (m'[i][y] = 0)
    }
  }
  check {
  }
}
```

2. **MR8**: Adds an exclusive lock to an unselected key

MR8 expression in DSL :

```
MR 8 {
  (select(k) where not contain(o,k))
  followup {
    (y' = y+1)
    (m' = addColumn(m))
    for (i,x){
      (m'[i][y] = 1 where i==k)
      (m'[i][y] = 0 where i!=k)
    }
  }
  check {
    (numKey' = numKey+1)
    (o' = add(o,k))
  }
}
```

An illustration of test-case generation for MR3 and MR8 are given in Table 2.

Detailed implementation of DSL specification for the Greedy problem, the generated input, output and checker are accessible here:

`https://github.com/ryanignatius/CheckerDSL/tree/master/Examples/Greedy/Greedy1`

6.4. *Other Case Studies from the Indonesian National Informatics Olympiad*

The same method has been used for test cases and checkers generator for some tasks in the Indonesian National Informatics Olympiad with MR corresponding to a DP (Dynamic Programming) solution. Complete definition of the tasks, MRs, test-case specification and the generated input, output, and checkers are accessible from:

`https://github.com/ryanignatius/CheckerDSL`

Table 2

Test-case generation example for MR3 and MR8 of a Greedy problem

Test Case	File Input	File Output	Explanation
Original	3 4 1 0 1 0 1 0 1 1 0 1 0 0	2 2 3	Original test case defined by problem setter. 1st line: the number of keys and the number of locks. 2nd line until the last line contains the definition of each key. Each number in each line defines the relation between a key and a lock.
MR3	3 5 1 0 1 0 1 1 0 1 1 1 0 1 0 0 1	2 2 3	File input and output are generated based on the original test case and MR3 (adding an insecure lock column)
MR8	3 5 1 0 1 0 1 1 0 1 1 0 0 1 0 0 0	3 1 2 3	File input and output are generated based on the original test case and MR8 (adding an exclusive lock to an unselected key)

7. Conclusion

A specifications-based test-case generator has been built for improving test-case generation and checkers. The generator has been used for Indonesian training-program task definition. The relation between input and output is checked by running the contestant's program. Whereas in the classical way checker is written to check one output run, in our system the checker is capable of checking the relation between two or more output executions, based on the Metamorphic Relation. Instead of writing a program, a problem setter writes a specification based on a DSL grammar. The specifications contain variables and their values, input-output format, input-output values, input-output constraints, score, MR and predicates representing a checker. The specifications will then be used to generate test cases and checkers. However, the usage of this system is not intuitive unless the problem setter has a minimum knowledge and understanding of MR and our DSL.

For a problem class, MR represents a property that can be reused in other similar problems in the same domain. We have proven the reusability of metamorphic relation for Knapsack, Greedy, DP, and numerical problems for generating test cases and checkers for an Indonesian national training task and IOI task. By using our method and tools, a problem setter can take advantage of previous experience and enrich the system.

The system also provides a library of predefined functions that will grow along with the experience of the problem setter. This generator will be useful for simple problems in which the input-output relationship can be expressed easily, even without writing solutions. This is the case in preliminary selection, such as national preparation where we have to conduct training programs with many simple tasks.

In this version, the code is generated in JAVA. In the future version, the generated code can also be applied to other languages, such as C, C++, or Pascal, by changing the DSL grammar.

References

Barus, A.C., Chen, T.Y., Grant, D., Kuo, F.C., Lau, M.F. (2011). Testing of heuristic methods: a case study of greedy algorithm. In: Zbigniew H. *et al.* (Eds.), *3rd IFIP TC 2 Central and East European Conference on Software Engineering Techniques (CEE-SET 2008), Brno, Czech Republic, 13–15 October 2008. (Lecture Notes in Computer Science, 4980)*. 246–260

Chen T.Y., Cheung, S.C., Yiu, S.M. (1998), *Metamorphic Testing: A New Approach for Generating Next Test Cases. Technical Report HKUST-CS98-01*. Hong Kong, Department of Computer Science, Hong Kong University of Science and Technology.

Chen, T.Y., Huang, D.H., Tse, T.H., Zhou, Z.Q. (2004). Case studies on the selection of useful relations in metamorphic testing. In: *Proceedings of the 4th Ibero-American Symposium on Software Engineering and Knowledge Engineering (JIISIC 2004)*. Madrid, Spain, 569–583

Ghosh, D. (2011). *DSLs in Action*. Manning Publication, 2011

Gotlieb, A., Botella, B. (2003). Automated metamorphic testing. In: *Computer Software and Applications Conference, 2003. COMPSAC 2003. Proceedings. 27th Annual International*

Im, K., Im, T., McGregor, J. D. (2008). Automating test case definition using a domain specific language. In: *ACM-SE 46 Proceedings of the 46th Annual Southeast Regional Conference on XX*. 180–185.

Mahmuda, A., Liu, H., Kuo, F.-C. (2011). On testing effectiveness of metamorphic relation: a case study. In: *Fifth International Conference on Secure Software Integration and Reliability Improvement, Jeju Island Korea, 2011.*

Mirzayanov, M. (2008). *Testlib*. `https://code.google.com/p/testlib`

Murphy, C. (2010). *Metamorphic Testing Techniques to Detect Defects in Applications without Test Oracles*. Columbia University Dept of Computer Science tech report cucs-010-10.

XText. (2014). `https://eclipse.org/Xtext`

Zhou, Z.Q., Huang, D.H., Tse, T.H., Yang, Z., Huang, H., Chen, T.Y. (2004). Metamorphic testing and its applications. In: *Proceedings of the 8th International Symposium on Future Software Technology (ISFST 2004)*. Japan, Software Engineers Association.

R.I. Hadiwijaya is a student in Informatics Engineering, Institut Teknologi Bandung, and an assistant in Programming Laboratory, Data & Software Engineering Research Group. He is doing his research in development of a systematic checker and test case generation for the automated grading system as a part of his final project, under supervision of Inggriani Liem.

M.M.I. Liem is a member of Data and Software Engineeing Research Group, School of Electrical and Engineering, Institut Teknologi Bandung (ITB). She has been teaching programming in ITB since 1977. She obtained her doctoral degree in Universite Joseph Fourier Grenoble France in 1989, with teaching programming as major topics of her dissertation. From 2004, she is involved as a team member in national recruitment, training and IOI preparation for Indonesian team. She is also ITB ACM ICPC coach and advisor.